There's an entire field dedicated to solving problems on strings. The book "Algorithms on Strings, Trees, and Sequences" by Dan Gusfield covers this field of research. Here are some sample problems:

- Given a text string and a pattern, find all occurrences of the pattern in the text. (Classic text search)
- The above problem where the pattern can have "don't cares" in it.
- Given a string, find longest string that occurs twice in it.
- Compute the "edit" distance between two strings, where various editing operations are defined, along with their costs.
- Given a set of strings, compute the cheapest tree that connects them all together (phylogeny tree)
- Compute the shortest "superstring" of a set of strings. That is the shortest string that contains all the given strings as a substring (important as a theoretical model in DNA sequencing).

Not only do these problems arise in our everyday lives (how do the `grep` and `diff` algorithms work?), but they also have many applications in computational biology. We're only going to scratch the surface of this field and talk about a couple of these problems and algorithms for them.

1 The Knuth-Morris-Pratt Algorithm

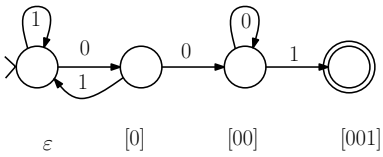
This algorithm can solve the classic text search problem in linear time in the length of the text string. (It can also be used for a variety of other string searching problems.) Formally, you have a pattern P of length p, and a text T of length t, and you want to find all locations i such that $T[i \dots i+p-1] = P$. For example, if you have a pattern $P =$ `ana` and $T =$ `banana`, then you would want to output locations $\{1, 3\}$, since $T[1 \dots 4] = T[3 \dots 6] =$ `ana`.[1]

An easy solution to this problem takes time $O(p \cdot t)$. This is fine if p is small, but as p gets large this becomes bad. *Can we do better?* It seems that if once we have checked whether $T[i \dots i+p-1]$ matches P or not, we have a lot of information that we can use to determine if $T[i+1 \dots i+p]$ gives a match. How do we do this?

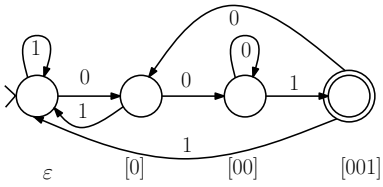
One high-level solution to this problem is to build a deterministic finite automaton M_P, which depends on the pattern. It consists of $p = |P|$ states, which you should think of as arranged in a line. We feed in the text T to this automaton. The properties of the DFA M ensure that we're in the j^{th} state if and only if at the current location in the string we've already matched a sequence of j characters from the pattern. Now we compare the next two characters. If we get a match we move to the next state in the list (matching $j+1$ characters). If we get a mismatch, we can skip back to some previous state. Which one? The start state of the automaton? Nope, that would be wrong. We go to the furthest back state that we know we can go to based on the information that we have.

For example, if the pattern $P =$ `001` and suppose we consider the DFA:

[1] We will assume that the strings are from some alphabet Σ. In some cases we will also assume that $|\Sigma|$ is a constant, we'll mention when we're doing so.

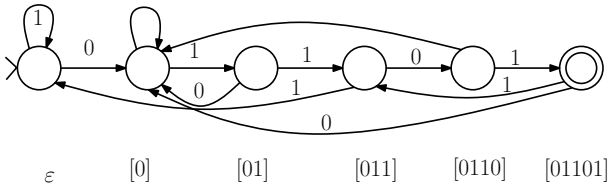


We would reach the final state (the one with the double circle) exactly when we have seen the pattern. This would allow us to find the first occurrence of pattern P in the text T. And in order to find all the occurrences of P, we could alter the DFA slightly to get this one:



We could output the current location in T every time we hit the final state, and then output all occurrences of P in the text T.

Another example: if $P = 01101$ then we'd build



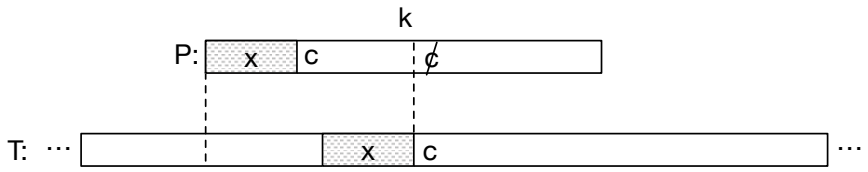
Clearly, once we have the DFA M_P, it takes just $O(t)$ time to feed the text T to the DFA, and record the locations in T when we reach the final state. If it takes us $f(P)$ time to build the DFA, the total run-time of the search algorithm would be $f(P) + O(t)$. It is easy to construct the DFA in time $O(p^3|\Sigma|)$, where recall that Σ is the alphabet. However, the algorithm of Knuth, Morris, and Pratt [2] suggests a way to compute the DFA in only $O(p|\Sigma|)$ time. The DFA is of size $O(p|\Sigma|)$, so this is optimal. In fact, their algorithm goes even further. While it is not presented as such, it can be viewed as building a related DFA-like automaton of size $O(p)$ (independent of the size of alphabet size Σ) that can be used for the string matching problem in $O(t)$ time.

1.1 A Fast Construction of a DFA

OK, so how did we really construct the DFA? For a pattern $P = s_0 s_1 s_2 \dots s_{p-1}$, let P_k denote the prefix $s_0 s_1 \dots s_k$. We have one state q_k for each prefix P_k. Suppose we are at state q_k. Now we want to see where to go when we see the character c after this. (Let $P_k \circ c$ denote the string $s_0 s_1 \dots s_k c$.) If $c = s_{k+1}$, then we go to the state q_k corresponding to $P_{k+1} = s_0 s_1 \dots s_k s_{k+1}$.

But suppose $c \neq s_{k+1}$. Where could the next match begin? Here's a picture of the situation after a mismatch at $k+1$:

[2] The historical notes in the original paper are interesting to read. Also, the *dramatis personae* may be familiar to you: Jim Morris is a professor of computer science here at Carnegie Mellon. He was Dean of the School of Computer Science from 1999-2004. Both Don Knuth and Vaughan Pratt are professors of computer science at Stanford. Don Knuth's books *The Art of Computer Programming* have been super-influential for the field, and he also developed the TEX typesetting system which has been used to create this document. He won the Turing Award in 1974. Vaughan Pratt was one of the authors of the deterministic median finding algorithm (along with Manuel Blum, Bob Floyd, Ron Rivest and Bob Tarjan); he also gave the proof you saw that PRIMES is in NP.



If a match were to start someplace in the region currently matched to P then:

- that match would start with a prefix of P (since all matches start that way), and
- that prefix would have to match in T up to k.

Such a string is marked as the shaded x region in the figure above. We want the *longest* such region so that we don't skip over a match in T. This means we need to find the *prefix* of P that corresponds to the longest *suffix* of P_k. That is, we need to find what is the fewest characters that we can drop from the beginning of P_k to get something that looks like a prefix of P again.

To see some examples of this, let us compute an array `memo[]` which records:

- `memo[k]`= the length of the longest *proper* suffix of P_k which is also a prefix of P.

Note we require a proper suffix, since we already know that the full suffix of P_k doesn't match.

Let's do an example. Suppose s is the string "`1110111101`". Here is `memo`:

```
  i        0   1   2   3   4   5   6   7   8   9
-----------------------------------------------------
memo[i]    0   1   2   0   1   2   3   3   4   5
```

Let's see how we got these numbers. Consider `memo[1]`:

```
s1..s1 =   1             (s0 is the first char of string)
s      =   1110111101
```

The match is of length 1 so `memo[1]`=1. All the suffixes we consider start at index 1 since we require proper suffixes.

```
s1..s2 =   11
s      =   1110111101
```

The match is of length 2 so `memo[2]`=2.

```
s1..s3 =   110
s      =      1110111101   (shifted to show no match)
```

No match so `memo[3]`=0

```
s1..s4 =   1101
s      =      1110111101
```

One match, so `memo[4]`=1.

```
s1..s5 =   11011
s      =      1110111101
```

Again a match so `memo[5]`=2.

```
s1..s6 =   110111
s      =      1110111101
```

Again a match so `memo[6]`=3.

```
s1..s7 =   1101111
s      =       1110111101
```

The length of the match is 3 so `memo[7]`=3.

```
s1..s8 =   11011110
s      =       1110111101
```

`memo[8]` = 4.

```
s1..s9 =   110111101
s      =       1110111101
```

`memo[9]` = 5.

Got it? Good.

1.2 Using the memo array for faster matching

The main idea is that `memo` gives us a compact representation of the DFA, except slightly simplified to have only 2 edge types: *match* and *mismatch*. *Match* edges just go from q_i to q_{i+1}, while `memo` gives the mismatch edges. Here's the pseudocode:

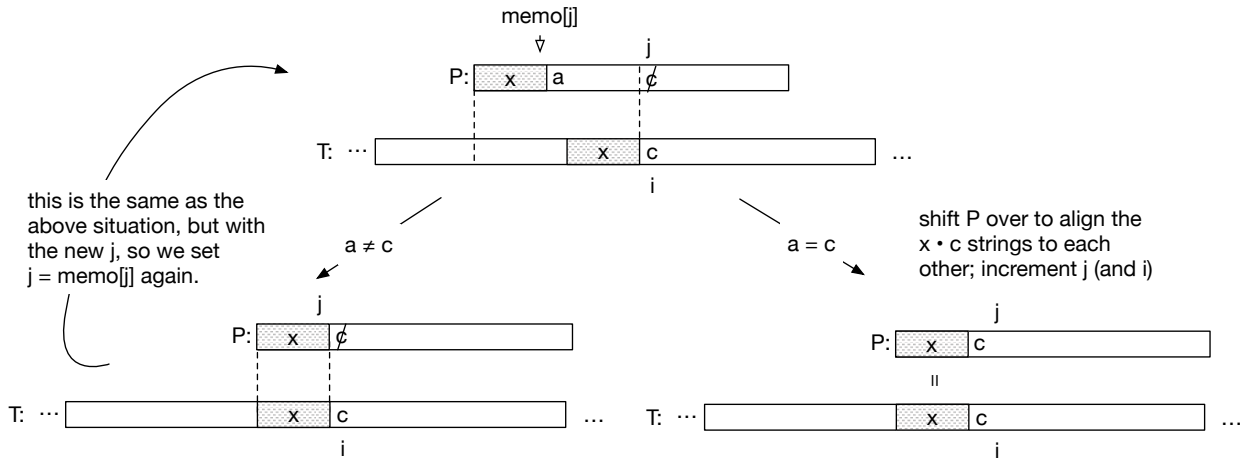
```
char[] T = "whatever text to search is";
// Assume strings are 1-indexed

int j = 0;                                          // position in pattern
for (int i=1; i <= T.length; i++) {
    while (j > 0 && T[i] != P[j+1]) j = memo[j];    // mismatch edge
    if (T[i] == P[j+1]) j++;                        // match edge
    if (j == n) {                                   // end state
        System.out.println("match found at "+(i-n+1));
        j = memo[j];
    }
}
```

Some notes:

- When $j = 0$ and we continue to mismatch, nothing in the body of the `for` loop will happen, and we will just be scanning `T` by incrementing `i`.
- Why do we execute `j = memo[j]` after we find a match? Because you can think of falling off the end of P as a mismatch between '\0' and T.

Here's a picture to help understand the `while` loop:



Why is the algorithm correct? It's simulating a simplified DFA for P. Each time there is a mismatch, P is shifted by the *least* amount for which a match could continue at i. So no matches will be missed.

How fast is it? The running time of the algorithm is linear. Each iteration of the inner 'while' loop decreases j (all the mismatch DFA edges point backward). Each iteration of the outer loop increases i and increases j by at most 1. Consider the quantity $q = 2i - j$. This increases for every bit of work the algorithm does: e.g.

- in the `while` loop, i is constant while j decreases $\implies$ q increases for each bit of work the while loop does.
- in the `for` loop if both i and j are incremented, q increases by 1. (This is why we use $2i$ instead of i in q.) If i is incremented, and j stays the same or decreases, then q increases by at least 2.

Finally, q is bounded between 0 (j can't get ahead of i) and $2m$ ($m \geq i, j \geq 0$). j will never get ahead of i since j is only incremented when i is.

1.3 Computing memo

Last step: How do we compute `memo` quickly? Because once it is computed, the actual matching takes time $O(|T|)$, if we can compute `memo` in time $O(|P|)$, we will achieve a total runtime of $O(|T| + |P|)$, which is linear in the input size. In fact, we can do this. Here's the code:

```
char[] P = "test"; // the pattern
// Assume strings are indexed starting from 1
```

```
int n = P.length;

int[] memo = new int[n+1];
/*  memo[i] will store the length of the longest prefix of P
    that matches the tail of P2...Pi */

int j=0;
for (int i=2; i <= n; i++) {
    while (j > 0 && P[i] != P[j+1]) j = memo[j]; // (*)
    if (P[i] == P[j+1]) j++;
    memo[i] = j;
}
```

This should look very familiar! It is essentially the match code we saw above but with $T = P$. Why should this make sense? When we are matching in T, we're always looking to extend the prefix of P that matches a tail of T. Using P in place of T does the same thing — we just have to record how long a prefix we matched (whereas in the match code, we only cared about full-length matches).

We again use the fact that j will never be ahead of i, so we will have already computed all the `memo[x]` that we need in the while loop for i.

The running time of computing `memo` is $O(n)$ by the same argument as for the match.

2 Boyer-Moore

Another algorithm for string matching that is often very good in practice is Boyer-Moore. We'll go over it at a high level so we can see some of its unique features. The most common version of it runs in $O(mn)$ time so it is not as good theoretically as KMP. But it often does much better than the worst case running time, and extensions exist to make it run in $O(m + n)$ time.

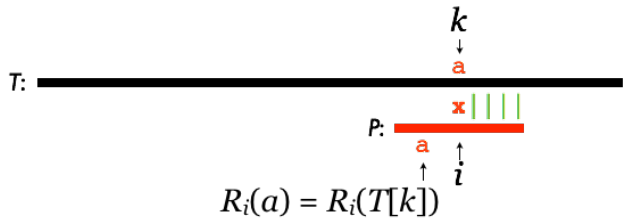
Idea #1: We will move the pattern P along text T from the left-to-right as before, but at each location where we put the pattern, we will compare P and T right-to-left.

Idea #2: Boyer-Moore will have two different rules for shifting the pattern by more than 1 character at a time. These heuristics will guarantee that no match will be missed, but they may not always apply.

First Shift Heuristic: Bad Character Rule. Define:

- $R_i(x)$ = position of the rightmost occurrence of character x before position i in the pattern P. So if $P =$ "abaaab", then $R_3(b) = 2$ and $R_3(a) = 1$.

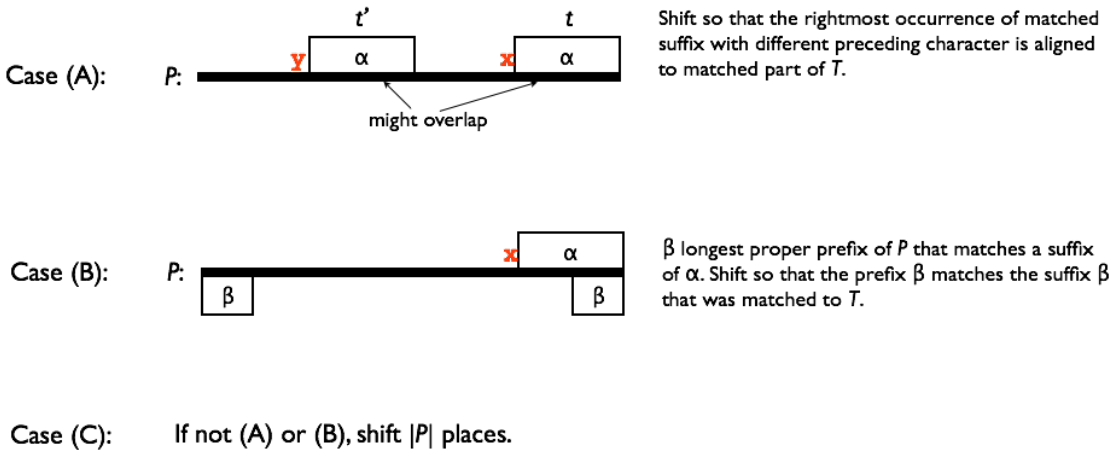
Bad Character Rule: When a mismatch occurs at pattern position i against k:



shift by $i - R_i(T[k])$ so that the next occurrence of $T[k]$ in the pattern is matched to position k in T. (This is called the "bad character rule" because it fires on a mismatch, but really it shifts so that the next *good* character matches.)

Second Shift Heuristic: Good Shift Rule. When a mismatch occurs, you will have matched some suffix α of P:

Apply these cases in order:



To handle these cases, we can pre-compute various features of the pattern, like we did with `memo[]` in KMP. Discussing how to do that would take longer than we have time for, unfortunately. However, Gusfield's book has a complete description, as does CLRS.

Complete algorithm: Putting these together, we have the Boyer-Moore algorithm:

```
k = 1
while k < |T| - |P| + 1:
    Compare P to T[k..k+|P|-1] from right to left. Stop at mismatch or complete match.
    if complete match:
        report match
        k += good shift rule
    else:
        k += max { bad character rule, good shift rule, 1 }
```